# Introduction

The idea is to predict the next possible location the user will visit. The dataset is taken from yelp dataset challenge available at <http://www.yelp.com/dataset_challenge> . Recommender systems have become a key tool for providing users with personalized recommendations on items such as movies, music, books, news and more. In today’s time, social influence plays an important role in product marketing or business. However, it has been rarely considered in traditional recommender systems. Tried to use the approach where user’s social influence (friends and friends of friends) is taken into account along with the traditional user preferences and item’s general acceptance while predicting new locations.

# Dataset Description

The dataset consists of the following information:

* Business
* Business Attributes
* Users
* Reviews
* Tips
* User Social Graph

Mainly used the business, user, review and user social graph information.

# Data Preprocessing

1. Used elasticsearch (an open source tool based on Apache Solr, <http://www.elasticsearch.org/>) to index data.
2. The data is provided in a single file.
3. Used the UNIX command line to break the file into smaller sub files.
4. The sub files where of mainly three types namely Business, User and Reviews. Each considering the specific information.
5. Used the pyes library based on python for elasticsearch for indexing and retrieval. The code files are present in the code folder.

Overview

The score for a location is calculated based on the following three factors:

1. User Preference: The users own preference for location based on his/her history
2. Item Preference: The overall preference for a particular location (which we are trying to predict) among all the users.
3. Friends Preference: The preference of the user’s friends for the location (which we are trying to predict). Set a threshold to require every pair of immediate friends to have at least three co-rated locations. If they do not, ignore their friend relationships.

A more detailed and comprehensive description of algorithm can be found at [1]

Inputuser = User for which new location recommendations are being made.

**Which locations to recommend?**

First find out the friends of the Inputuser. Combine all the locations reviewed by the friends (say X). Remove the locations which are already visited by the Inputuser from X. The list of the remaining locations are the places which the Inputuser might visit. Hence, that will be the set of recommendations for the Inputuser. One point to note here is that, if there are no friends, currently the algorithm will fail to predict any locations.

**How to calculate everything at run time?**

This is a social graph and some users may have many friends and hence large number of locations to predict. In such scenario, to predict everything at the run time takes a huge amount of time for the score calculations. So preprocessed and stored the following information on files (which is common for all the runs)

1. **User to Business Dictionary**: For each user, store all the business he has reviewed.

Key = user\_id

Value = list of business\_id

1. **User to Business Rating Dictionary**: For each user, store all the ratings he has given to the business which he has reviewed.

Key = user\_id

Value = dictionary (key = business\_id, value = rating given by user)

1. **Business to User Dictionary:** For each business, store all the users who have visited that business.

Key = business\_id

Value = list of user\_id

1. **Business Acceptance Rating:** For each business, calculate and store its common acceptance across all users

Key = business\_id

Value = Array [totalCount, k1, k2, k3, k4, k5]

ki = number of ratings received where star = i. For example; if k3 = 10, this means 10 users have given rating as 3 to this business. TotalCount = k1+k2+k3+k4+k5

1. **Price Attribute Dictionary:** For each business, if the price attribute is given then store it.

Key = business\_id

Value = price attribute (ranges from 1 to 4)

**Recommendations based on categories**

For a given Inputuser, if predicting score for locations irrespective of categories, then the number of locations to predict becomes quite large (in thousands). This does not make much sense because for recommendations we are generally interested in top k items. Also users generally search locations based on their area of interests. So there is a variable as inputCategory, which will filter the locations to recommend based on user input. It is set to “Restaurants” by default.

**Recommendations based on city**

Initially tried to recommend places across available cities even if the user has not been to that city. This gave some poor results as it highly unlikely that user residing in city A, will be visiting locations of city B. So applied another filter on locations to recommend based on city. First see how many cities the Inputuser has visited and then recommend places in those city only.

**Recommendations based on Immediate Friends**

In this approach, consider the social influence only up to the immediate friends on Inputuser. For example, if A has immediate friend as B and B has immediate friend as C, then in no way we consider C’s opinion about a location to calculate score for A.

**Recommendations based on Friends Immediate Friends**

In this approach, extend the social influence one step ahead from the immediate friends of Inputuser. For example, if A has immediate as B and B has immediate friend as C, then we also take into account C’s opinion about a location to calculate score for A.

**Evaluation of Results**

Used a simple metric to calculate top 10 and top 15 items. For the given Inputuser and for the inputCategory, we know the restaurants/locations the user has visited (say n). Using the immediate friends approach try to score all the possible restaurants/locations within the particular city (of inputCategory) (say x). Rank them in descending order as the score are numeric units. Then check, out of ‘n’, how many locations are present in ‘x’ in top 10 or top 15 positions (say y). Then the result metric is y/x. In ideal case scenario, y/x should be equal to 1 quantifying that all the n locations are present in top 10 or top 15 positions.

**Results Overview:**

1. The results are better if we try to predict locations for a single city rather than multiple cities.
2. Currently input users are selected with a prior condition that it should have at least one immediate friend.
3. For better understanding of user behavior, input users are selected which have rated/visited at least 10 locations.

**Steps to reproduce:**

1. Start the elasticsearch
2. Run the files in code/index folder (sequential/bulk) to index the documents.
3. Run the files in code/preprocessing folder to create dictionaries for future use.
4. Run the snrs\_immediate\_predict.py file to predict the locations
5. Run the code/evaluation/snrs\_Immediate\_eval.py file evaluate the metrics.

**Future Improvements**

1. Even if there are no immediate friends for an Inputuser, the algorithm should still be able to predict
2. While indexing documents, it is much better if documents are store in a single (lower case). Consider a term for eg: ‘AbC’. It is indexed as ‘abc’ but is stored as ‘AbC’. So while search
3. If we give ‘AbC’, nothing is fetched.
4. If we give ‘abc’, ‘AbC’ is fetched.

To make things simple, everything should be in lower case. Store, index and search as ‘abc’. This can be done while indexing documents. Right now used default settings.

1. Can try to use map reduce functionality to reduce the time and avoid storing of dictionaries on files.
2. The approach of including price attribute and distant friends can be explored more. Ideally it should give more consistent results than immediate friends approach as this is an additional information used to predict the score.
3. Along with price attributes, other business attributes can be also be experimented.

**Code Information:**

1. Three following python(.py) files for three approaches

snrs\_immediate\_predict.py

snrs\_immediate\_price\_predict.py

snrs\_distant\_predict.py

1. code/index folder contains two files related to indexing
2. code/preprocessing contains files related to preprocessing stuff that will be used in actual approaches
3. code/evaluation contains three files with respect to three different approach for evaluation
4. code/scripts contains simple python supporting scripts

**References:**

[1] <http://www.cobase.cs.ucla.edu/tech-docs/jmhek/snrs.pdf>

[2] <http://www.elasticsearch.org/>

[3] <https://pypi.python.org/pypi/pyes/>

[4] <http://www.stackoverflow.com>